Parametrising a theory of software problem solving

Conference or Workshop Item

Parameterising a theory of software problem solving

Jon G. Hall
Dept. of Computing and Communications
The Open University
Milton Keynes, UK
Jon.Hall@open.ac.uk

Lucia Rapanotti
Dept. of Computing and Communications
The Open University
Milton Keynes, UK
Lucia.Rapanotti@open.ac.uk

ABSTRACT

We explore what can be said about the detailed modelling of problem solving ability via a stochastic semantics of a theory of software problem solving, and end with an invitation to discuss possible experiments that may lead to the practical characterisation of the problem solving ability of software teams.

1. INTRODUCTION

In [**?**] we proposed an initial model for software development process modelling using Problem Oriented Engineering (POE, [**?**]). In particular, we identified the *problem solving unit of composition* based on the POE Process Pattern (PPP, [**?**]), shown in Figure 1, and used it to provide process architectures for both waterfall and agile processes. In this short paper, we exploit further characteristics of that problem solving unit to say more about the modelling of teams' problem solving ability.

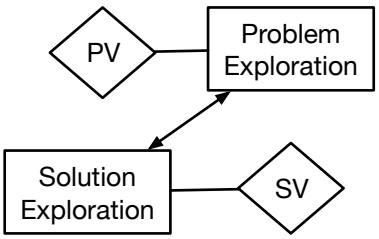

Figure 1: The POE Process Pattern

To this end, we propose a stochastic semantics of the PPP and perform a simple experiment. Our desire is to inspire discussion on the possibilities of running such experiments in the local communities of participants at the workshop.

2. BACKGROUND

TOSE'16, May 16 2016, Austin, TX, USA

ISBN 978-1-4503-4174-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897134.2897137

In Design research (for instance, [**?**]), design is considered as the fundamental process through which real-world problems are addressed via the invention, assembly and adaptation of technologies, which involves planning and decision making, as well as the concrete realisation of artefacts. POE is an emerging problem solving approach to engineering, the creative, iterative and often open-ended undertaking of designing and constructing products, systems and processes that address real-world problems.

POE is *design theoretic* [**?**], by which we mean it provides a theory that characterises the elements of problem solving in terms of the effect they have on the process of design, and only thus on an artefact[1]. Problem solving in POE includes many types of design activity, including Weick's 'sensemaking' [**?**], various formal and informal refinement techniques, Jackson's problem progression [**?**], architectures [**?**], *etc.*, each characterised by its effect on the design process.

A more complete description of POE than can be included in this position paper (including its relationship to the design and other literature) can be found [**?**].

3. SOFTWARE ENGINEERING AS PROBLEM SOLVING

POE specialises Rogers' definition of engineering [**?**] to software engineering as:

> "Software engineering refers to the practice of organising the design and construction of any software system which transforms the physical world around us to meet some recognised need."

In this way, software engineering becomes a problem solving exercise, the problem being, given a real-world environment E, to find the software C that satisfies a real-world need N to the satisfaction of a group of stakeholders K, written $\mathsf{E}(\mathsf{C})$ $\mathrm{meets}_{\mathsf{K}}$ N. We assume that E contains a computational engine — a PC, mainframe, cloud service or the like.

POE characterises problem solving as the interaction of two processes [**?**]:

(i) one operating in the problem space, which contains the context E and needs N, and where problem solvers liaise with the 'problem owner' stakeholders to understand and describe and then validate their problem;

(ii) one operating in the solution space, which contains the solution C, and where problems solvers liaise with

[1]Design theoretic is defined by analogy to Gentzen's Proof Theoretic approach [**?**].

'solution owner' stakeholders to understand and describe their needs of the solution.

Interleaved with these are stakeholder validation points: problem owners validate problem descriptions ('PV' in the figure); solution owners validate solution descriptions ('SV in the figure').

Accordingly, we can identify three problem solving states for each space:

(i) the state before a validatable understanding exists, written $\mathsf{P}\times$ and $\mathsf{S}\times$, respectively;

(ii) the state when a validatable understanding exists, but before it has been validated, written $\mathsf{P}?$ and $\mathsf{S}?$;

(iii) the state when validated understanding exists, written $\mathsf{P}\surd$ and $\mathsf{S}\surd$.

As described, the result is two independent processes. However, the processes are not independent as solution exploration should not commence before a validated problem is uncovered[2] and the process of validation of a solution may reveal that the problem (even if it had been validated) was incorrect. Moreover, there are many forces working against their composition being simple, for instance: the reduced availability of problem owners to give their validation of the problem understanding when the need arises; the tendency or need to move into the solution domain before a validated understanding of the problem is gained; the volatility in the context, meaning that previously validated understanding is rendered out of date; that resources aren't sufficient to solve the problem, forcing premature exit; *etc.*

The resulting process is complex and iterative; further details, together with a fuller discussion of the literature that is possible in this short article, can be found in [**?**].

A validated solution to a validated problem, i.e., software that has, for instance, passed its acceptance tests, constitutes a solved problem and so is an end state in the process. The other end state is when the problem has been accepted as being without solution, within the available resources.

4. A STOCHASTIC SEMANTICS OF THE PPP

Discrete-time Markov chains (DTMCs) model memoryless stochastic processes, i.e., processes whose development is governed by random variables and whose future development is determined only by its current state and not by the sequence of events that preceded it. Given our assumption that a problem solving team is characterised by the likelihood of problem or solution validation success after exploration, we can model both problem and solution processes as Markov processes and analyse long-term behaviour. We can use the PRISM model checking tool [**?**] for process simulation and analysis. PRISM allows 'experiments' to be run on a parametrised model by varying those parameters.

An initial PRISM encoding of the process is shown in Figure 3. The listing shows *constant and formulæ definitions* (those without initial values can be varied in an *experiment*), four process *modules* and a *reward clause*, explained next.

[2]Of course, this does not discount the possibility that work in the solution space might be a useful part of problem exploration, and *vice versa*.

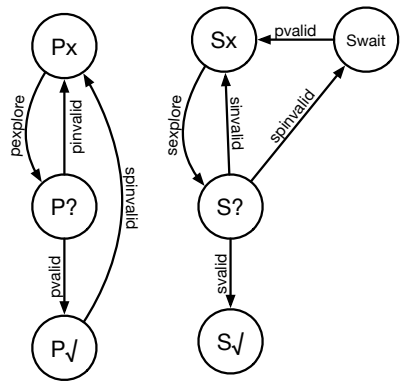

Figure 2: Discrete Time Markov Chains for (left) Problem and (right) Solution processes.

Constant and formulæ definitions. These include: (i) stamina, which models that problem solving is a resource intensive activity[3]; (ii) pexpertise (resp., sexpertise) the likelihood that any one request for problem (resp., solution) validation will be successful. These characterise team expertise in the sense that more expert teams will be more successful when they validate with stakeholders. (iii) svalidbias which measures how likely solution validation reveals that the problem description isn't actually valid, i.e., the problem validator made a mistake; (iv) the ok formula captures that energy remains; and (v) the solved formula characterises that the successful end state has been reached, i.e., the problem is solved.

Module problem encodes the problem process shown on the left of Figure 2 as a DTMC.

Module solution encodes the solution process shown on the right of Figure 2 as a DTMC.

We note that, due to limitations of the PRISM modelling language, certain auxiliary states and transitions are needed to model non-determinism in the processes of Figure 2, with the result that some transition labels shown in Figure 2 are omitted in the listing.

Modules problem and solution interact through two transitions:

- pvalid, by which the solution process is constrained to wait until the the problem process is validated complete, and
- spinvalid, by which the problem process is reset should validation of the solution reveal that the 'wrong problem has been solved'.

Module activityMonitor is an auxiliary (deterministic) process to monitor resource levels. We thus model that software engineering is resource constrained.

Module solvedMonitor is an auxiliary (deterministic) process to monitor whether the problem has been solved or when the teams run out of energy. The problem is solved when both problem and solution have reached their respective end states together. The problem is unsolved when resources are exhausted.

[3]Without stamina all (solvable) problems would eventually be solved.

```
dtmc

const int stamina=5;//min 3

const double pexpertise=17/(17+13);//EARLY
const double sexpertise=6/(6+21+3);//EARLY
const double svalidbias=21/(21+4);//EARLY

formula ok = energy>0;
formula solved = s=4;

module problem
        p : [-1..3] init 0;

        [pexplore]      p=0 & ok -> (p'=1);
        [pvalidate]     p=1 & ok -> (1-pexpertise): (p'=0) + pexpertise: (p'=2);
        [pvalid]        p=2 & ok -> (p'=3);
        [spinvalid]     p=3 & ok -> (p'=0);
endmodule

module solution
        s : [-1..6] init 0;

        [pvalid]        s=0 & ok -> (s'=1);//waiting for validated problem understanding
        [sexplore]      s=1 & ok -> (s'=2);
        [svalidate]     s=2 & ok -> (1-sexpertise): (s'=3) + sexpertise: (s'=4);
        [s]             s=3 & ok -> (1-svalidbias): (s'=5) + svalidbias: (s'=6);
        [sinvalid]      s=5 & ok -> (s'=1);
        [spinvalid]     s=6 & ok -> (s'=0);
endmodule

module activityMonitor
        energy : [-1..stamina] init stamina;

        [pexplore]      energy > 0 -> (energy'=energy-1);
        [sexplore]      energy > 0 -> (energy'=energy-1);
endmodule

module solvedMonitor
        svd : bool init false;

        [solved]        solved -> (svd' = true);
        [giveup]        energy=0 -> (svd'=false);
endmodule

rewards "cost"
        [pexplore] true: 1;
        [sexplore] true: 1;
endrewards
```

Figure 3: A stochastic semantics for the POE problem solving pattern

The **rewards** clause, separately from the activityMonitor, counts all time taking steps (problem and solution exploration) which allows cumulative costs to be recorded.

5. OUR EXPERIMENT

We wished to exercise our model in a parsimonious way on an easily accessible class of problems. The problems had to be representative, accessible and illustrative with skills extant for their solving.

Ever since their use to identify those with the particular problem solving skills needed to crack German encryption at Bletchley Park during WWII ([**?**]), cryptic crosswords have held an affectionate place in UK problem solving life.

Thus, the problem class that we chose is cryptic crosswords. Although cryptic crosswords may appear someway distant from those encountered in software engineering — their scale and subject matter being very different, for instance — we would argue that they have some of the characteristics of problems in that they have both a problem (the cryptic clue itself) and solution understanding part; moreover, unlike mathematical problems, problem and solution can be ambiguous. Also, the link between crosswords and software is not entirely unknown: Apt [**?**] relates the solving of crosswords to constraint programming.

Cryptic crossword clues are, in the following sense, self-validating: a cryptic clue (the problem) is constrained to be grammatical and has three components, a *definition*, a *puzzle* and a *letter count*; its solution is a word or phrase. The definition can be arbitrarily long and, as the name suggests, defines the solution. The puzzle, the remainder, is a grammatical word play that cleverly tells the solver how the solution should be built. The complex relationship between clue and solution leads to work in the problem space — in gaining validated understanding the clue — and solution space — constructing a validated answer — as well as iteration between the two — where a valid but incorrect understanding of the clue is revealed by exploration of the solution space.

We do not claim that the complexity of cryptic crossword solving rivals to any extent that of anything but the simplest of software engineering tasks. Rather, we claim only that they both represent problem solving activities that match the problem solving patterns that are available in POE.

	Prob Valn		Soln Valn		
Phase	Succ	Uns	Succ	Uns	Prob Inv
Early	17	13	6	21	3
Middle	17	12	13	16	2
Late	6	20	4	3	0

Table 1: Results of the observed crossword problem solving process

5.1 Solving a cryptic clue

Generally, the first step in solving a cryptic clue is to identify definition and puzzle parts. As definition and puzzle are contiguous, they form either a pre- or suffix of the clue. That they must make grammatical sense individually is a form of self-validation (i.e., validation without recourse to the setter).

An example of a cryptic clue is:

> "Welcome alternate dignitary to top middle-of-road club, it's said (12)"

Here the definition is 'Welcome'[4]; the puzzle is the remainder. The solution, having 12 letters, is 'Introduction,' a word meaning welcome, which is defined by the puzzle in the following way: take 'alternate' letters of 'dignitary' — i-n-t-r — prefixed 'to' the 'middle of' 'top' — o — followed by a synonym of 'road' — d-u-c-t — followed by a homonym ('it's said') of a type of (golf) 'club' — i-o-n..

A simple experiment is, then, to observe a crossword problem solver to identify the various problem solving phases, and to parametrise the stochastic model accordingly.

We identified a cryptic crossword from Private Eye magazine (#564, [**?**]). The first author was solver. From his experience of solving cryptic crosswords, we determined that there would be three phases to the experiment: Early (when no or very few intersection letters exist), Middle (when some intersection letters exist) and Late (when almost all intersection letters exist). For each phase we assigned an hour, broken down into 60 minute intervals. We gave each attempt to solve a clue 5 minutes. At the end of each minute we recorded the state of problem solving process in terms of $*\times$, $*?$ and $*\checkmark$ (where $* \in \{\mathsf{P}, \mathsf{S}\}$). When a clue was solved we moved to another without waiting.

6. FINDINGS

The results of the three phases are shown in Table 1; the first line (Early) should be read as, in the first hour, we observed 30 successful Problem Validation events (i.e., the solver had found what they thought was a valid definition), 17 of which were successful, and 30 Solution Validation events (i.e., the solver had found the answer), 6 of which were

[4]But it could have been a longer clue prefix or suffix.

successful (i.e., a valid solution was found), with 3 of those unsuccessful indicating that the clue's definition was invalid too.

The results were instantiated in the model as constant probabilities (see Figure 3). The resulting DTMC was 'verified' through the PRISM tool. We ran the following queries:

- `P=?[F solved]`: to give the probability (`P=?`) of successful solving[5]. PRISM returned `p` = 0.236 (to 3 d.p.) suggesting the expected solved number of 4/16, which compares to the actual of 6/16;
- `R=?[F solved]`[6]: to give the expected cost (`R=?`) of problem solving, when successful. PRISM returned `c` = 2.844.

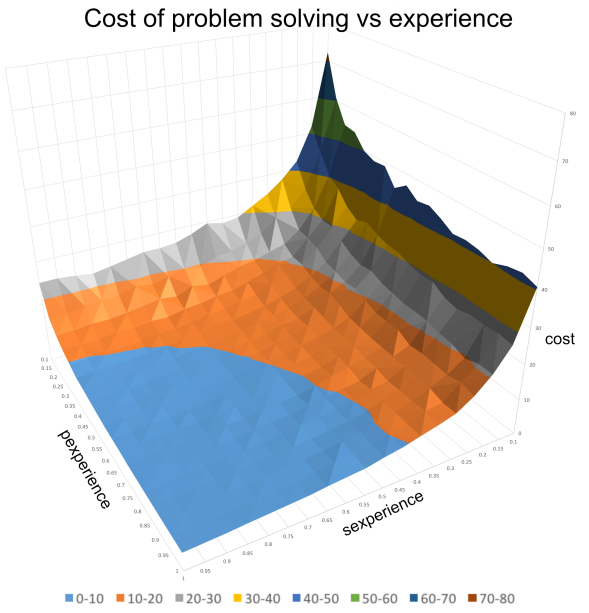

Figure 4: A 3D plot of cost versus solution and problem experience for a problem solving experiment similar to that described, but with effectively unconstrained resources: (sexperience, pexperience $= 0.1...1.0$ **step** 0.05**) with the vertical representing the total cost of solving (**min $= 0$, max $= 80$**). The asymmetry between problem and solution is due to the `spinvalid` transition.**

7. CONCLUSIONS AND FURTHER WORK

We have given an initial DTMC semantics for the PPP and shown how observations of a problem solver in action can be used to parametrise it. Other possible metrics we could investigate under such semantics include, for instance, problem solving risk, total cost of solving, *etc.*

Work is in progress on a compositional semantics of PPP under a simple process-algebra [**?**] which, in combination with the ideas in [**?**], may allow us to encode complex software development processes in terms of the PPP, thus providing a stochastic semantics based, so making them susceptible to (a more sophisticated form of) the analysis of this paper.

Acknowledgements

Thanks are due to Dariusz Kaminski for his contributions to the ideas expressed here.

[5]From the initial state, what's the probability (`P=?`) of eventually reaching (`F`) a state in which `solved` holds.

[6]For technical reasons, the actual query was `R=?[F "deadlock"]-stamina*P=?[F energy=0])/P=?[F solved]`.